SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **10**, 111–150 (1998)

Research

Debugging Program Failure Exhibited by Voluminous Data

TAT W. CHAN[1] and ARUN LAKHOTIA[2]*

[1]*Department of Mathematics and Computer Science, Fayetteville State University, Fayetteville NC 28301, U.S.A.*
[2]*The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette LA 70504, U.S.A.*

SUMMARY

It is difficult to debug a program when the data set that causes it to fail is large (or voluminous). The cues that may help in locating the fault are obscured by the large amount of information that is generated from processing the data set. Clearly, a smaller data set which exhibits the same failure should lead to the diagnosis of the fault more quickly than the initial, large data set. We term such a smaller data set a *data slice* and the process of creating it *data slicing*.

The problem of creating a data slice is undecidable. In this paper, we investigate four generate-and-test heuristics for deriving a smaller data set that reproduces the failure exhibited by a large data set. The four heuristics are: invariance analysis, origin tracking, random elimination and program-specific heuristics. We also provide a classification of programs based upon a certain relationship between their input and output. This classification may be used to choose an appropriate heuristic in a given debugging scenario. As evidence from a database of debugging anecdotes at the Open University, U.K., debugging failures exhibited by large data sets require inordinate amounts of time. Our data slicing techniques would significantly reduce the effort required in such scenarios. © 1998 John Wiley & Sons, Ltd.

KEY WORDS: data slicing; fault reproduction; origin tracking; heuristics for slicing; slicing effectiveness

1. INTRODUCTION

To find the cause of a failure, i.e., debugging, a programmer often observes the values of program variables—the state—at various intermediate steps during execution. This is done by exercising the program through a data set or a sequence of events exhibiting the failure. For most programs, the number of intermediate steps and the amount of information in the state at each step depends upon the size of the failure data. A large data set may

* Correspondence to: Arun Lakhotia, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette LA 70504, U.S.A. E-mail: arun@cacs.usl.edu

Contract/grant sponsor: Louisiana Board of Regents; Contract/grant number: LEQSF (1993–95) RD-A-38
Contract/grant sponsor: U.S. Army Research Office; Contract/grant number: DAAH 04-94-G-0334

CCC 1040–550X/98/020111–40$17.50

Received 20 December 1995
Revised 28 September 1997

cause failure after a large number of iterations, thereby generating a large number of intermediate steps. Moreover, the size of the intermediate values of variables (such as arrays) may be proportional to the size of the input, thereby generating a large amount of information in each state. Such debugging scenarios may arise even when the program has been exhaustively tested using one or more testing criteria (be it branch coverage, mutation coverage or dataflow coverage). During actual use one may create data or events that exercise conditions not exercised by the data in the test suite. That the failure-causing data set is large may just be a matter of chance.

Debugging a program when the failure is *exhibited* by a large data set is hard because the clues that may help in finding a fault are obscured by the large amount of information a programmer has to process. Clearly, a smaller data set reproducing the same failure would ease the debugging. Creating such data set is not always easy. It is especially difficult when the program has been tested extensively, because then one has to identify the specific conditions not exercised by data in the test suite—a task that may require finding clues from the vast amount of state information. Creating a smaller, failure-reproducing data set is also not easy when a program's input is generated by another program or preprocessor—as happens in applications, such as image processing, statistical analysis and compiler backends. In such cases one must create a data set such that the preprocessor outputs a smaller failure-reproducing data set, a task that further adds to the complexity of the problem. The alternative is to isolate the program and manually construct its input. This can be an equally complex task if the program's input has a complex structure.

This relationship between *debugging effort* and the *choice of data* used was first observed in our earlier work (Chan and Lakhotia, 1993; Lakhotia and Chan, 1993). In this paper, we further expand on the preliminary ideas presented earlier and present a collection of methods to 'create a smaller data set that reproduces a failure *exhibited* by a *voluminous* data set'. By the phrase 'large' or 'voluminous' data set, we do not only mean that the size of the input data set is large. An input data set is voluminous if processing it requires an enormous amount of computation—thereby creating a large number of internal states. A data set may also be voluminous if the amount of information in the internal states is large. The phrase 'exhibited by' may be understood by contrasting it with the phrase 'caused by'. A failure 'exhibited by' a large data set may be reproduced by a smaller data set. In contrast a failure 'caused by' a large data set implies that the failure is not reproducible by any smaller data set.

We term a 'smaller' data set that reproduces a failure exhibited by a large data set a *data slice*. The process of creating data slices is called *data slicing*. The term 'slice' is intended to draw an analogy with Weiser's definition of a *program slice* (Weiser, 1984; Chen and Cheung, 1997). A program slice is a 'smaller' program—made up of statements contained in a given program—preserving some syntactic and semantic constraints. The analogy between the two is that the 'smaller' artefact is created from the elements of a given artefact while preserving certain semantic constraints. A program slice is not just a 'smaller' program that preserves certain semantic constraints, it is also a program constructed from the statements of a given program. Similarly, a data slice is not just a smaller data set that produces the same failure as the original data set, it is also constructed from the elements of this data set.

Chan, the first author, has proposed a framework for debugging that incorporates the use of data slicing along with program slicing and traditional trace-and-inspection tools (Chan, 1997). A summary of the data slicing techniques presented in this paper is also available in Chan's paper.

The rest of the paper is organized as follows. In Section 2, we further motivate the discussion by means of a case study and the results of two informal surveys. In Section 3, we present the constraints influencing the design of our data slicing methods. In Section 4, we give (i) a classification of programs and data domains, and (ii) terminology of a failure, same failure and a fault. These classifications are helpful in choosing the appropriate data slicing method in a particular debugging scenario. Section 5 presents several methods for data slicing. In Section 6, the effectiveness of our data slicing methods is assessed. Section 7 surveys various debugging techniques and explains why they are unable to help in debugging failures exhibited by voluminous data. Section 8 contains our concluding remarks.

2. MOTIVATION

2.1. Case study

The following case study illustrates the difficulty of debugging a failure exhibited by a voluminous data set and how a data slice helps in overcoming the difficulty.

We were once developing a program to recover the architecture of a program based on the approach suggested by Hutchens and Basili (1985). This program uses *cluster analysis* to organize functions of a C program in a hierarchy. It takes as input a $n \times n$ matrix where N is the number of functions in the given program. The matrix contains the 'binding' strength between functions, the number of *global variables*, *typedefs*, *macros* and symbols used by both the functions, etc. The output of the program is an *n-ary* tree that can be represented as a binary tree with at most $N-1$ interior nodes. The program contains a main loop. In each iteration of the main loop, m elements are clustered into a node of the *n-ary* tree and the dimension of the matrix is decreased by $(m-1)$.

We tested the program on several sample matrices. Once satisfied with its correctness, we started using it in our research. After some time, we observed a failure when a 209×209 matrix was used as input. In order to debug, we used *dbx* to set breakpoints and to inspect the variables at various points in the execution. Unfortunately, the amount of information output was overwhelming and would rarely fit on a screen. Figure 1(a), for instance, shows what a typical screen looked like. It was cluttered with numbers. Since the program failed after 20 iterations, debugging the program by tracing its execution was impossible. We did, however, try tracing the program for two days but were not even able to detect any clues that would allow the debugging to proceed. We then used invariance analysis and random elimination techniques (discussed later in this paper) for creating data slices and were able to create a 7×7 matrix, see Figure 1(b), that also produced the same failure as the original data. Subsequently, we were able to debug the program in just two hours, since the clues—relationships between various elements of the matrix—were easily visible from looking at the intermediate matrices. The creation of

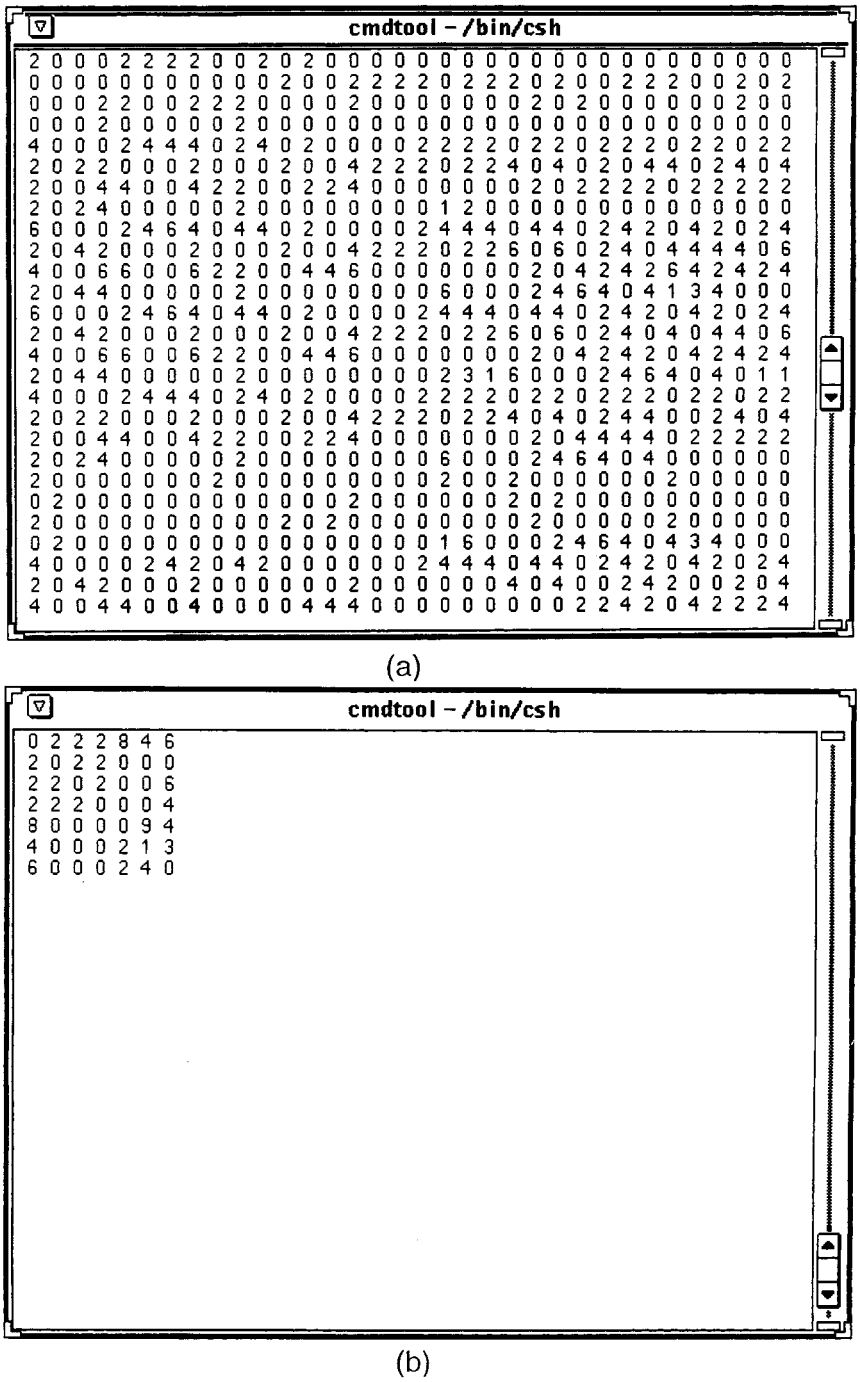

Figure 1. The effect of data slicing. The screen (a) shows only a part of the actual 209 × 209 matrix. The matrix in screen (b) was created from the 209 × 209 matrix using data slicing techniques proposed in the paper

the data slice itself required about five hours, including the time for developing some special code to support the data slicing techniques.

2.2. Results of two informal surveys

Scenarios similar to the above case study have been experienced by other programmers too. The database of debugging anecdotes maintained at the Open University in the U.K. contained (as of January 1994) a collection of 163 anecdotes (Eisenstadt, 1993). Of these, 12 anecdotes narrate scenarios where the failures were exhibited by large data sets (or sequence of events). The effort required to debug, reported in each of these anecdotes, ranges in weeks and months. There are two extreme cases where the debugging itself was terminated.

The difficulty of debugging in these 12 anecdotes is attributed either to (a) the amount of information generated by the program for the failure-causing data set or (b) the vast amount of time required to reproduce the failure.

That debugging under the stated scenario is difficult is further corroborated by the responses to our query, see Figure 2, posted on the Internet newsgroups *comp.bugs.misc*, *comp.source.bugs* and *sci.image.processing*.

Some typical responses we received were:

'When debugging an image processing algorithm, conventional debuggers are of little help. The vast amount of data that an image generates makes it fruitless to examine each number.'

'This sort of bugs ARE difficult to track down.'

'Your work is definitely interesting, and a real problem in debugging ... Several of our SERC affiliates have had problems with programs that ran for a long time or generated lots of output before failures.'

My dissertation work is focussing on some techniques for debugging failures exhibited by voluminous data . . . [explanation of the problem omitted] . . . I would like to assess how significant the problem is and how people deal with it. Would you please comment on the following questions?

1. Do you know of or have experienced debugging situations in which the failures were exhibited by large data?

If yes,

2. Can you briefly describe the experiences?

3. How difficult was it to debug the failures in (1)?

4. How frequently have you encountered such failures?

Figure 2. Query posted on the Internet newsgroups comp.bugs.misc, comp.source.bugs *and* sci.image.processing *to survey the difficulty of debugging failures exhibited by large data*

Although the above surveys were not controlled, they do indicate that it is extremely difficult to debug a program when the data exhibiting the failure is large. Clearly, methods that aid in creating a smaller data set that reproduces a failure would help in reducing the debugging effort and sometimes even enable the debugging to proceed when it may be impossible otherwise. Such methods are, therefore, worth research investigation. Details of our above surveys are documented elsewhere (Chan, 1994).

3. CONSTRAINTS ON THE DESIGN OF DATA SLICING METHODS

The problem of determining whether two data sets input to a program will produce the same output is undecidable, in the general case. Similarly, the problem of generating another input data set that produces the same output as a given data set, for an arbitrary program, is also undecidable. The above problems may be generalized by changing the constraint 'produce the same output' to 'satisfy the same assertion'. Since a failure may be expressed as the violation of an assertion, see Section 4, the problem of finding another data set that reproduces a failure exhibited by a given data set, for an arbitrary program, is also undecidable. Hence, the problem does not have any algorithmic solution and must rely on searching the total solution space. Appropriately, therefore, in response to our query of Figure 2 we received the following suggestions for creating a data slice:

'. . . I suggest two techniques:

- Start from a small data set and gradually expand and complicate it until it breaks.
- Take the set that doesn't work and simplify or shrink it until you can obtain a better understanding of what is going wrong.'

These are essentially generate-and-test search methods, wherein first a data set is chosen as a candidate for the solution and then it is tested to verify whether indeed it is a solution. In the first technique suggested, a candidate data slice is created without making use of the data set causing the failure, whereas the second technique starts from the original data set. The heuristics we provide belong to the second class and are based on the hypothesis that:

Hypothesis: *When a failure is found in a given program with an input data set* ***d****, if there exists a smaller data set which reproduces that failure, it can be created from* ***d****.*

While this hypothesis may not always hold, in our experience, the searches using our heuristics converge very rapidly, see Section 6, when the hypothesis does hold.

In all search algorithms, no single heuristic is guaranteed to find a solution in all situations. The same is true with data slicing, as demonstrated by the following example.

Consider a program *Rowmax* which processes a given matrix and outputs a list of values. Each value in the expected output list is a maximum of a row in the matrix. For program *Rowmax*, each output value is only 'related' to the values in one row of the input matrix because only those values are used to compute the *row maximum*. Assume that in an execution of program *Rowmax*, one value in the list of maximums is wrong.

If the data slicing method *M* creates a data slice using all the input values that are 'related' to the erroneous output values, then it will find a data slice which is the right row in the original input matrix.

While method *M* is good for performing a data slice for program *Romax*, it may not be good for another program. For example, consider a program *Determ* that computes the determinant of a given matrix. The determinant is 'related' to all the elements in the input matrix because all the elements are *used* to compute the determinant. If the determinant computed by program *Determ* for a given matrix is wrong, using method *M*, the data slice obtained may be the whole input. Using this method, therefore, one can never get a data slice for program *P*.

We, therefore, present a collection of heuristics for creating data slices. We further observe that the choice of the appropriate heuristics in a debugging situation may be guided by the input and output domains, the program itself, the type of failure and the fault causing the failure. In the next section, we provide a classification scheme for data domains and programs. This classification may be used in choosing the appropriate heuristics.

4. DATA, PROGRAMS, FAILURES AND FAULTS

In this section we classify data domains and programs. We also give definitions for a failure, same failure and a fault.

4.1. Data

In this subsection, three aspects of data will be discussed: the data domains, data elements and the size of a data set.

4.1.1. Data domains

The input and output data domains can be classified into three categories:

- primitive types, such as integers and characters;
- fixed structures, such as fixed length arrays and records of all types; and
- recursive structures, such as lists, trees, graphs and variable length arrays of all types.

Inputs and outputs of any program can be modelled using these categories since programs, in the most general sense, are simply transformers of data.

4.1.2. Data elements

The fixed and recursive structures, when decomposed completely, consist of values of primitive types. A primitive value of a data structure is called its data element. A datum of a primitive type has only one data element, the datum itself. The values of primitive data types contained in a fixed or recursive structure are its data elements. For example, the primitive values in each of a linked list are its data elements. Similarly, the data

elements of a tree consist of the primitive values that comprise its nodes. If data values were also associated with the links of the linked list (or the edges of a tree), then the associated primitive values are data elements too.

4.1.3. Size of a data set

So far the word 'smaller' has been used intuitively. A measure for a data set that quantifies the difficulty of debugging a program with a given data set is needed. A data set requiring greater debugging effort should have a larger value for this measure. Such a measure is called *data size*, or just *size*.

Like any metric, size may be defined in many ways. The most obvious measure of size is storage size.

Definition: storage size. *The number of bits required to store the data in memory is called storage size.*

The storage size of an input is usually a good indicator of the number of intermediate states required to process it. For instance, a factorial program would perform more computations to compute the factorial for 100 than for seven. The storage size for 100 is seven bits and is greater than the storage size of four bits for 10.

However, storage size is not always a good indicator of the intermediate states and hence of the debugging effort. Consider, for example, a program which reads a list of integers denoting consecutive leap years from the year 4 A.D. and generates the remaining leap years up to the 400th leap year. Then an input list containing the first 10 leap years will require more processing than the list containing the first 20 leap years. But the storage size for 20 years is greater than the storage size for 10 years.

Clearly, the debugging effort does not depend solely upon the storage size of the input data set; it also depends on the program being debugged.

Definition: processing size. *The amount of computation performed by a program to process a given input gives the processing size for the input.*

Given a program and a data set, the processing size of the data set may be computed by instrumenting the program to get the number of the times a loop (implicit or explicit) is iterated and the depth of recursive function calls. (Count for statements that are not in a loop or recursive function are not significant since (a) if they are not in a loop or recursive function they can be executed at most one time and (b) if they are in a loop or recursive function then they are executed as many times as the loop is executed.) Test coverage analysis tools, such as *btool* (Hoch, Marick and Schafer, 1990), Asset, and ATAC (Horgan and Mathur, 1992), can provide such coverage information. In the absence of a tool, it may be estimated analytically by using the computational complexity of the program and the storage size of the data set. Besides the number of intermediate states, the size of each state—the amount of information at each state—also has an influence on the debugging effort. It has been our experience that the size of the intermediate states for a data set tend to be proportional to its processing size. While this is not conclusive,

it suggests that the processing size is a good indicator of the debugging effort, since it is proportional to both the complexity of the computation and the storage size of the data set.

4.2. Programs

4.2.1. Definition of programs

A program is typically defined as a collection of code (in some programming language) that can be compiled into object code and executed on a machine. We use a more general definition. A program can be a single statement or a composition of statements. A statement may be an assignment statement, an input statement, an output statement, an alternation statement or a repetition statement. This general definition is consistent with how a programmer views a program during debugging. Instead of treating it as a monolithic piece of code, one uses a divide-and-conquer strategy to identify suspect modules or sets of statements. This strategy is most evident when the program being debugged contains a succession of phases in which data are transformed. If a failure is found in a particular phase, the debugging effort may be focused on that phase only. In such cases, if an intermediate phase p is being debugged, then the data slicing methods may create smaller data for that phase alone, not necessarily the whole program. Notice that since phase p is an intermediate phase, its input is the values of all variables which are (a) read by the input statements in phase p or (b) referenced before modification in phase p.

4.2.2. Program specification

We use Morgan's notation for specification of program: $pre \Rightarrow P\ post$ (Morgan, 1988), where condition *pre* is the weakest precondition (Dijkstra, 1976) of program P with respect to a postcondition *post*. The specification denotes that if program P is activated in a state for which condition *pre* holds, then it will terminate in a state for which *post* holds.

4.2.3. Classification of programs

The specification of a program gives the relations established between its input and its output. The relations given by the specifications should capture the complete behaviour of a program. These relations may, however, be abstracted to express only the relations established between the primitive elements of the input and output such that:

(a) With every element e_i in the input, one can associate a set $\pi(e_i)$ in the output such that e_i 'generates' the set of elements $\pi(e_i)$.
(b) With every element e_o in the output, one can associated a set $\mu(e_o)$ in the input such that element e_o is 'generated from' the set of elements $\mu(e_o)$.

The relationship between an input element e_i and $\pi(e_i)$ may be classified as follows:

- Type $\alpha_1 : \pi(e_i) = \phi$, i.e., e_i does not generate any element in the output.
- Type $\alpha_2 : \pi(e_i) = \{e_i\}$, i.e., e_i generates the element e_i itself in the output.
- Type $\alpha_3 : \pi(e_i) = \{e_o\}$, i.e., $e_i \neq e_o$, i.e., e_i generates a single element e_o in the output, where e_o is different from e_i.
- Type $\alpha_4 : |\pi(e_i)| > 1$, i.e., e_i generates more than one element in the output.

The relationship between an output element e_o and $\mu(e_o)$ may be classified as follows:

- Type $\beta_1 : \mu(e_o) = \phi$, i.e., e_o is not generated from any element in the input.
- Type $\beta_2 : \mu(e_o) = \{e_o\}$, i.e., e_o is generated from the element e_o in the input.
- Type $\beta_3 : \mu(e_o) = \{e_i\}$, $e_i \neq e_o$ i.e., e_o is generated from a single element e_i in the input, where e_i is not the same as e_o.
- Type $\beta_4 : |\mu(e_o)| > 1$, i.e., e_o is generated from more than one element in the input.

While these relationships can be derived from a program's specification, in practice, a complete written specification may not be necessary. Given an element in the input (or output), a programmer may be able to intuitively *classify* its type—α_i (or β_i), $1 \leqslant i \leqslant 4$—from the knowledge of its expected behaviour.

4.3. Failures and faults

4.3.1. Definition of failures

A failure may be defined as any deviation of a program's behaviour from the expected behaviour. For example, if a program 'hangs' when it is expected to terminate, then it is a failure. However, if a program is not expected to terminate—such as an operating system—then its non-termination is not a failure.

When first observed, a failure is associated with some external behaviour of a program. As debugging proceeds, the failure gets associated with one or more subcomponents of the program. In the process, the definition of the failure itself changes and becomes more precise. For instance, an initial observation that a program does not terminate on debugging may be associated with non-termination of a loop. In this case, the failure may be that the termination condition of the loop cannot be established (for the given input).

This successive translation of failure may be captured using program specifications. The specification of a program (and subprograms) defines assertions that are expected to hold at various program points. A failure is a violation of an assertion. The violation of an assertion at a subcomponent may result in the violation of assertions for its parent component, therefore leading to an externally observable failure.

4.3.2. Same failure

Let a program P be specifed as $pre \Rightarrow P\ post$, where *pre* is its precondition and *post* is its postcondition.

The program P is said to fail on a data set d that satisfies the condition *pre* if the program fails to establish the postcondition *post* when invoked with the data set d.

Two data sets d_1 and d_2 cause the *same failure* if they both violate the same subconditions *post′* of *post*, i.e., *post′* $\Rightarrow$ *post*, d_1 causes violations of *post′* iff d_2 causes violation of post′.

4.3.3. Definition of faults

A fault is also commonly known as a bug, which is the condition in the program which causes a failure. A fault is, therefore, manifested by a failure. A failure implies that there must be a fault in the program but a fault may not necessarily cause a failure.

5. DATA SLICING METHODS

Definition: data slice. *Given a data set d_1 that causes a program p to fail, data set d_2 is a* data slice *of d_1 iff*

(1) *d_2 reproduces in p the same failure as d_1 and*
(2) *d_2 is smaller than d_1 when compared using processing size with respect to program p.*

Data slicing is the method of creating d_2 given program p and data d_1. In this section, five methods for data slicing are discussed. As stated in Section 3, the methods are based upon the hypothesis that: *when a failure is found in a given program with input data* ***d****, if there exists smaller data which reproduces the failure, it can be created from* ***d****.*

To debug failures exhibited by a large data set, one would first create a data slice using one or more of the data slicing methods, and then use a general purpose debugging technique to debug the program.

The rest of this section discusses the following data slicing methods:

(1) Data slicing by invariance analysis.
(2) Data slicing by origin tracking.
(3) Data slicing by random elimination.
(4) Data slicing by program-specific heuristics.

5.1. Data slicing by invariance analysis

Data slicing by invariance analysis is especially designed to create data slices for loops in a program. Consider the situation where some computation in a loop fails after 50 iterations, where the failure may be defined as a violation of its loop invariant. When tracing the program using debuggers such as *dbx*, one would have to step through the code 50 times before coming to the state when the program fails. The invariance analysis method helps in creating a data slice that would cause the loop to fail after a single iteration.

Figure 3 gives the steps for data slicing by invariance analysis. This method has the following formal basis. Let M_0 denote the state just before entering the loop M. In successive iteration, M_0 is transformed to $M_1, M_2, \ldots M_n$. If the loop invariance is violated

1. Develop an invariance for the failing loop

2. Immediately after the entry in the loop, introduce code that does the following:

```
If loop invariance is established
then
    Save the state of all varible used or modified in the loop and the loop predicate
else
    Report failure
    Output the data slice - the state of the variables saved in the last iteration
```

Figure 3. Steps for creating a data slice using invariance analysis

after the kth iteration, then the state M_{k-1} will be a *data slice* for the loop. If we use M_{k-1} as the entry state for the loop, its invariance will be violated immediately after completing the first iteration. $M_1, M_2, \ldots M_{k-2}$ are also data slices of M_0 but not as *small* as M_{k-1} (Figure 3).

5.1.1. Example using invariance analysis

The program in Figure 4 is intended to compute the sum of the 10 elements in the array b. It has a fault. The operator '*' is used instead of '+'. From its specification, the invariance of the loop is expected to be $\sum_{k=1}^{i-1} b[k]$, $0 < i < 11$. Suppose that the initial input to the program is:

$$\text{Array } b\text{: } 0,0,0,0,5,6,7,8,9,10$$
$$i = 1,\ sum = 0$$

The invariance holds at the beginning of the first five iterations, i.e., with $1 \leqslant i \leqslant 4$.

```
program sum-array
...
  sum :=0
  i :=1
  while(i < 11) do
    sum := sum * b[i] // error sum := sum + b[i] //
    i := i+1
  end_while
```

Figure 4. A program to sum the elements of any array. The program is faulty in that it computes the product. The operator '' is being used instead of '+'*

It violates at the beginning of the sixth iteration. If the loop was re-executed with values of *sum* and *i* as 0 and 5, respectively, the loop invariance will be violated immediately after the first iteration. These values along with the array *b* (not changed in the loop) form a data slice for the loop.

Notice that although the original input is for the whole program, the data slice is input only to the loop.

Reverting the state of the variables to the last iteration before the invariance is violated can easily be done with a debugger such as Spyder (Agrawal, DeMillo and Spafford, 1991). Thus, using invariance analysis, Spyder may be used for creating limited types of data slices.

While this method may create a data slice that violates a loop after the first iteration, it does not always reduce the size of the data significantly. For instance, in the case study given in Section 2.1 invariance analysis helped in creating a 198×198 matrix from a 209×209 matrix. The smaller matrix still had too much information to be useful in debugging.

Furthermore, the applicability of this method depends upon (a) whether the programmer knows or can derive the loop invariance, (b) the support available for instrumenting the program to detect the violation of the invariance, and (c) the computational cost for checking the violation of the invariant.

5.2. Data slicing by origin tracking

The term 'origin' is borrowed from van Deursen, Kling and Tip (1993). The *origin* of an element in the output is the input elements which 'cause' these elements to be generated. The union of the origins of all the faulty elements in the output is termed the *failure origin*. The failure origin can be used to create a data slice. The faulty values in the output are dependent on the *failure origin* with respect to the program. We have investigated three methods to locate the origin based on (a) *dynamic program slicing*, (b) *abstract interpretation*, and (c) *input–output analysis*. The process of tracking the origin is illustrated in Figure 5. First, the erroneous output elements are identified in the output

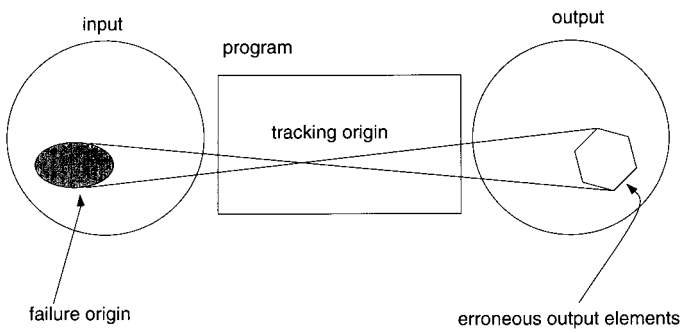

Figure 5. Illustration of tracking the origin. Origin tracking can be done by dynamic program slicing or abstract interpretation

of a program, then by applying one of the origin tracking methods, the failure is asssociated to some of the data elements in the input.

A failure origin itself is not a data slice; it is only a set of elements that constitute a part of a valid input. The set of all valid inputs to a program is its input domain. In other words, the origin may not be in the input domain of the program. Therefore, after obtaining the origin, we try to create a data set that belongs to the input domain of the program and contains the origin so that it is more likely to be a data slice. Such a data set is a candidate data slice. The methods to compute *candidate data slices* depend on the domain of the input. For a fixed structure, the candidate data slices can be obtained by 'filler elements substitution'—replace the elements not in the origin by some 'small' elements. Data slicing by origin tracking is most rewarding if the failure-causing data set is a recursive structure. However, more careful consideration is needed for a recursive structure when obtaining candidate data slices from the *origin*. This is because the positions of the erroneous elements in the failure origin may be significant to reproduce the failure. These issues will be discussed in Section 5.2.4. The following summarizes the steps to perform data slicing by origin tracking:

(1) Track the failure origin using:
   - *dynamic program slicing*, or
   - *abstract interpretation*, or
   - *input–output analysis*.

(2) Create a set of candidate data slices such that each candidate is:
   - smaller than the original data set and
   - contains the failure origin.

(3) Test each candidate to see if it reproduces the failure.

In the following, we will describe how the origin may be obtained by dynamic program slicing, abstract interpretation and input–output analysis. The methods for selecting candidate data slices are described after that.

5.2.1. Origin tracking by dynamic program slicing

Agrawal and Horgan's algorithm for computing a dynamic program slice (Agrawal and Horgan, 1990) is influenced by Horwitz, Prins and Reps' algorithm for computing the static program slice (Horwitz, Prins and Reps, 1988). The latter slicing algorithm represents a program as a program dependence graph (PDG)—a graph denoting the essential control and data dependences between statements of a program. A static program slice over this graph for a given statement, say *n*, consists of all the statements that can reach the node corresponding to *n* in the PDG.

Agrawal and Horgan's algorithm maintains a dynamic PDG in which every invocation of a statement is represented by a new vertex and retains only those dependences which are actually manifested during an execution. Therefore, computing the dynamic slice in this graph is also a graph reachability problem.

The origin of a set of faulty elements may be obtained by dynamic slicing using the following steps:

(1) Scan the output of the program to identify parts (or elements) that are faulty. This step is performed by the programmer.
(2) Trace the faulty elements in the output back to the statement instance in the dynamic PDG that output it. If the output is a hard copy or a file, this may be a manual task. If the output is on the screen, this mapping may be maintained by the system such that a pointing device (e.g., a mouse) may be used to query it.
(3) Compute a dynamic program slice over the set of statement instances identified to have output faulty elements.
(4) Identify the input statement instances in the dynamic program slice. The data elements input by these statements is the failure origin for the faulty elements of the output.

A more detailed and formal description of the algorithm for origin tracking by dynamic program slicing may be found in Chan (1994).

5.2.2. Origin tracking by abstract interpretation

Computing the origin by dynamic program slicing relies on maintaining a dependency graph during execution. The origin is obtained by traversing this graph *backwards*. The origin of some output elements can be computed by extending the semantics of the language in which the program is written. This is an abstract interpretation and is a *forward* approach to obtain the origin.

The principle of the method is to propagate the dependencies of the input elements, during execution, to other variables in the program. Both data and control dependencies are propagated. The propagation is modelled by changing the standard semantics of the language. In the standard semantics, a variable is associated with a value. The semantics are extended by associating each variable with, in addition to a value, a set of *tags* indicating the transitive data dependencies and control dependencies on elements of the input data set. The tag is a two-tuple, (<line-number>, <instance number>). The first field gives the line number of a *read* statement and the second field gives the execution instance of that statement. In the rest of the document, the tag will be represented as $<\text{line-number}>^{<\text{instance number}>}$. For instance, if a read statement on line 2 is executed the third time, then the tag created will be represented by 2^3.

For each faulty element in the output, the origin can be obtained from the set of tags associated with them. The following explains in more detail how the semantics of a simple structured language with *read*, *assignment*, *if-then-else*, *do-while*, and *write* constructs are extended:

(1) At the beginning, the tag set for each variable is made empty. The *control dependency set* is empty too. The control dependency set contains tags associated with variables used in predicates in the program.
(2) When an *assignment* statement is executed, then

(a) if the expression on the right-hand side contains only constants, then the set of tags associated with the variable on the left-hand side will contain tags in the control dependency set or
(b) if the expression on the right-hand side contains at least one variable, then the variable on the left-hand side is associated with the union of:
(i) the sets of tags, each set of which is associated with a variable used on the right-hand side and
(ii) the set of tags in the control dependency set.

(3) When a *read* statement is executed, a new tag is created and associated with the input variable, along with the tags in the control dependency set.
(4) Upon entry to an *if* statement, all the tags associated with the variables in the *predicate* are added to the control dependency set.
(5) Upon exit from an *if* statement, all the tags added upon its entry are taken out.
(6) Upon each iteration of a *do-while* loop, all the tags used in the *predicate* are added to the control dependency set.
(7) Upon exit from a *do-while* loop, all the tags added upon its entry are taken out.

To compute the origin of a faulty output element, one has to:

(1) Identify the set of tags associated with the variable containing the element.
(2) In that set of tags, identify the tags which represent *read* statement instances.
(3) The set of values read at the *read* statement instances is the origin.

The origin of a set of faulty output elements is the union of the origins of each faulty output element.

A more formal treatment of the concept of the abstract interpretation and an example of its application to find the origin is given in Chan (1994).

5.2.3. Origin tracking by input–output analysis

Tracking the origin by dynamic program slicing and abstract interpretation is computationally expensive. Moreover, if the failure is some 'missing output', the failure origin cannot be tracked by either of these two methods. In these cases, data slices may be created by input–output analysis.

Origin tracking by input–output analysis creates candidate data slices based on the relationship between input and output data elements derived from the specification. The advantage of this method is that the program code need not be explored to derive a candidate data slice. Therefore, it can be performed quite quickly.

Origin tracking by input–output analysis makes use of the relationships between the input and output elements to create a data slice. The following are two cases where input–output analyses are recommended:

(1) Output elements of type β_2 missing.

From the specifications or expected behaviour, it is determined that an element e_0 of

type β_2 (see Section 4.2.3) is missing from the output, then e_0 (being identical to an element in the input) belongs to the failure origin. Origin tracking by dynamic slicing or by abstract interpretation cannot be applied in this situation since there are no faulty output elements in the output. In Section 6, where an assessment of the data slicing methods is presented, Case Study 1 presents an example of this case.

(2) Failure related to an input element of type α_4.

From the specification if it is determined that a wrong output value is 'generated from' an input element e_i which is of Type α_4 and $|\pi(e_i)|$ is large, then one can derive a candidate data slice by replacing e_i with e'_i such that $|\pi(e'_i)|$ is 'smaller' and still contains the wrong output value. However, the derivation of e'_i without exploring the code is possible only when it is derivable from the specification. Case Study 3 in Section 6 presents an example of this case.

The applicability of input–output analysis is limited by how much we can infer about the relationship between input and output elements from the specification.

5.2.4. Creating data slices from failure origin

A failure origin identifies the elements of an input data set contributing to the creation of faulty elements in the output. A failure origin itself is not a data slice; it is only a set of elements that constitute a part of a valid input. The set of all valid inputs to a program is termed its *input domain*. Therefore, the next problem is to create an input to the program using the elements in the origin such that the input would produce the original failure. If the input domain can be represented by a language, say L, the problem may be reduced to creating a string $d \in L$ consisting of elements in the failure origin. There are two causes of concern:

(a) It may not always be possible to create a valid string using elements from the failure origin alone. Figure 6 shows an example where the input domain is any square matrix, the failure origin consists of three elements and the failure origin cannot be a valid input.

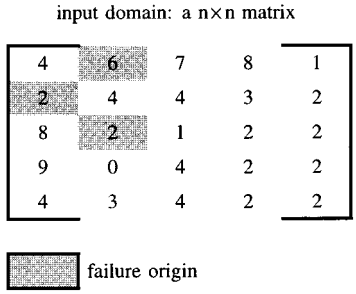

Figure 6. Relationship between a failure origin and a data slice. This example shows that the failure origin alone may not be a data slice because it may not be a valid input

(b) When it is possible to create a valid string using the failure origin, the resulting string may still not be a data slice.

The first problem can be solved by using some 'filler' elements along with the failure origin to create a syntactically valid string. Ensuring that the resulting string will be a data slice is a harder problem. In the following, we discuss, for each input domain, how one may consider creating data slices after obtaining the *failure origin*.

5.2.4.1. Input domain is of primitive type If the input domain consists solely of elements of a primitive type such as integers, the best way to create a data slice is by *invariance analysis*. Dynamic slicing does not reduce the input data set since the whole input data would be in the failure origin.

Thus, for a factorial program that fails for input 5, the failure origin would be {5}.

5.2.4.2. Input domain is of fixed structure Fixed length arrays, records and matrices with fixed dimensions are termed *fixed structures*. If a program processes a fixed structure, say a 5×5 matrix, then a data slice for it would also be a 5×5 matrix, even if the origin consists of only a few elements from the matrix. The only real candidate for a data slice would be a replica of the input with all the elements not in the origin replaced by some 'small' element of the primitive type of the structure. Thus, for a 5×5 matrix of integers causing errors in a program that processes only 5×5 matrices, the 5×5 matrix in which all elements of the failure origin appear as they do in the original matrix and all other elements are 'filled' with 0 would be a likely candidate for a data slice. Other candidates may be chosen by using a number other than 0 for 'filling the holes'. The example in Figure 7 illustrates '6', '2' and '2' are elements in the failure origin. We obtain a candidate by replacing all other elements by '0'. We cannot have any matrix of smaller dimensions since the input domain is a fixed structure.

5.2.4.3. Input domain is a recursive structure Using origin tracking for data slicing is most rewarding when the input domain consists of recursive structures such as sequences, trees and matrices of variable dimensions. Using the origin, a set of smaller structures

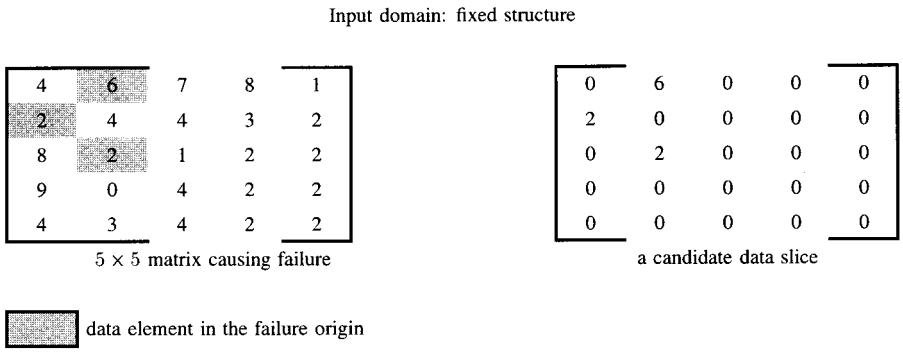

Figure 7. Creating data slices for a fixed structure. An example of data slicing by origin tracking for an input domain with fixed structure. Given that the input domain consists of 5×5 matrices and a 5×5 matrix causing a failure, one can only create a candidate data slice by filling the elements not in the failure origin with some 'small' elements

can be heuristically generated such that one of its elements may be a data slice. This set is generated based on the following classification of relationships between a failure and the failure origin.

- *Absolute position dependent*: the failure depends upon the position of the elements of the failure origin in the input data.
- *Relative position dependent*: the failure depends upon the relative position of elements within the failure origin.
- *Order dependent*: the failure depends upon the ordering of elements in the failure origin.
- *Content dependent*: the failure depends upon just the collection of elements in their failure origin.

Notice that the relationships are presented in decreasing order of strength, i.e., a relationship subsumes all subsequent relationships. Thus, if a failure depends upon the absolute position of the elements of the failure origin, it also depends upon the relative position, order and content. A set of candidate data slices may be created by creating data items that contain all the elements of the failure origin and also preserve the absolute position relation, relative position relation or order relation between them. The stronger the relationship a data item preserves, the greater chance it has of being a data slice (due to the subsumption of relations). However, such a data item would also be larger, hence of less value in debugging, than the data items preserving the weaker relationships. The aim is to find the smallest of these items that is also a data slice. Some of the candidate data items may be ignored using program specific knowledge. For the remaining data items, whether they are data slices may be determined by testing the program with them as input. This testing may be performed in the order of the size of the data items.

Sequences: If the input domain consists of sequences, the set of sequences that may be considered for data slices, based on the above rules, would be:

(1) The sequences consisting of elements in origin ordered as in the original input sequence (order dependent).
(2) The smallest subsequence of the original data containing all elements of the origin (relative position dependent).
(3) The smallest prefix of the original data containing all elements of the origin (absolute position dependent).
(4) The smallest suffix of the original data containing all elements of the origin (absolute position dependent).
(5) A sequence generated using techniques for fixed structures (absolute position dependent).

The first sequence would be a data slice if the error depends only upon the elements in the origin and their relative order. The second sequence would be a data slice if the relative positions of these elements are important too. If the absolute position of these elements from the beginning (or end) of the original sequence influences the failure then the third (or fourth) sequence would be a data slice. If the absolute positions from both

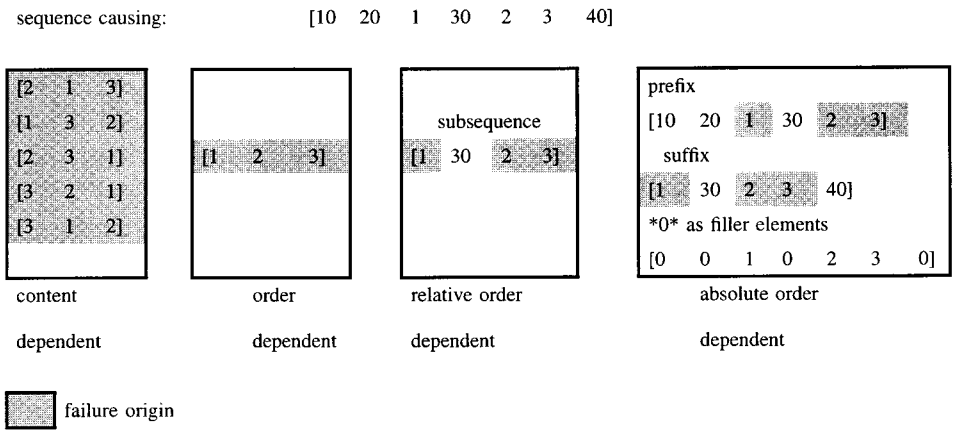

Figure 8. Positioning of elements in failure origin and data slicing. Given a sequence causing a failure and the failure origin the reproduction of the failure may depend on the different ordering of the elements in the failure origin

ends are relevant then one cannot create a data slice of length smaller than the original failure causing sequence. The data slice may still, however, be constructed by using techniques suggested for fixed structures. Figure 8 illustrates the above definitions. Figure 9 gives an example program in which the smallest prefix as per Item 3, above, is a data slice. The program fails for input (0, 2, 3, 4). The corresponding failure origin is {3} and candidates for data slice {(3), (0, 2, 3), (3, 4) (0, 0, 3, 0}. Of these, only (0, 2, 3) and (0, 0, 3, 0) are data slices, since the error depends on the (only) element in the origin being at the fixed position 3 starting from the front of the sequence, i.e., its smallest prefix.

Binary trees: To define rules for creating a set of candidate data slices for binary trees we first need to define the notion of order, relative position and absolute position in a

```
begin
 read (n)
 i := 1
 while (i <=n) do
   read (a[i])
 end
 r :=(1)**a[3]      /*    error: r ← (−1)^a[3] */
 write(r)
end
```

Figure 9. Creating data slices for sequences. The program fails for input (0, 2, 3, 4). The corresponding failure origin is {3} and candidates for data slices are {(3,), (0, 2, 3), (3, 4), (0, 0, 3, 0)}. Of these, only (0, 2, 3) and (0, 0, 3, 0) are data slices since the error depends on the (only) element in the origin being at the fixed position 3 starting from the front of the sequence, i.e., its smallest prefix

binary tree. The ordering of elements of a tree may be defined based upon the strategy used for traversing its elements. The traversal strategies may be broadly classified as: depth-first and breadth-first. Amongst these there may be further classifications, for instance, infix, prefix, or postfix depth-first traversal, or left-to-right, or right-to-left breadth first traversal.

We use the following for defining the relative and absolute positions.

- *root*: the root node of the original tree.
- *nca(origin)*: the nearest common ancestor of all the elements in an *origin*.
- *nca-subtree(origin)*: the subtree rooted at the *nca(origin)*.
- *neck(origin)*: the branch i.e., nodes and edges, on the path from the *root* to *nca(origin)*, both inclusive.
- *top(origin)*: the tree consisting of *nca-subtree(origin)* and *neck(origin)*.
- *left(origin)*: the tree consisting of *top(origin)* and all the subtrees rooted on the left branches of nodes in *neck(origin)*.
- *right(origin)*: the tree consisting of *top(origin)* and all the subtrees rooted on the right branches of nodes in *neck(origin)*.

To avoid clutter, the original input tree is left implicit and not specified as a parameter. Figure 10 illustrates the above definition.

The *nca(origin)* preserves the relative positions between the elements of the failure origin, for any type of depth-first traversal. Therefore, this tree is used in creating relative position dependent candidates for a data slice.

The absolute positions of the elements of the failure origin may be defined relative to (a) the root of the original tree, or (b) its left-most leaf, or (c) its right-most leaf. An example of a program failure that depends upon absolute position from the root is a failure that depends on an element appearing at a specific depth in a tree. Similarly, a failure may depend upon an absolute position from the left or right if the failure is caused by an element occurring at a specific position in a depth-first or breadth-first traversal. The subtrees *top(origin)*, *left(origin)*, and *right(origin)*, respectively, preserve these absolute positions and may be used to create candidate data slices.

The set of candidates for data slices for a tree therefore consists of:

(1) The smallest tree containing elements from the origin ordered on the traversal strategy used by the program (order dependent).
(2) The *nca-subtree(origin)* (relative position dependent).
(3) The *top(origin)* (absolute position dependent).
(4) The *left(origin)* and *right(origin)* subtrees (absolute position dependent).
(5) The trees created by treating the original tree as a fixed structure (absolute position dependent).

The first subtree is too dependent upon the traversal strategy. In most circumstances one may first create the *nca-subtree*, Item 2 above, since it satisfies the requirements for the first. From the second to the fifth subtree the size of a candidate subtree increases as

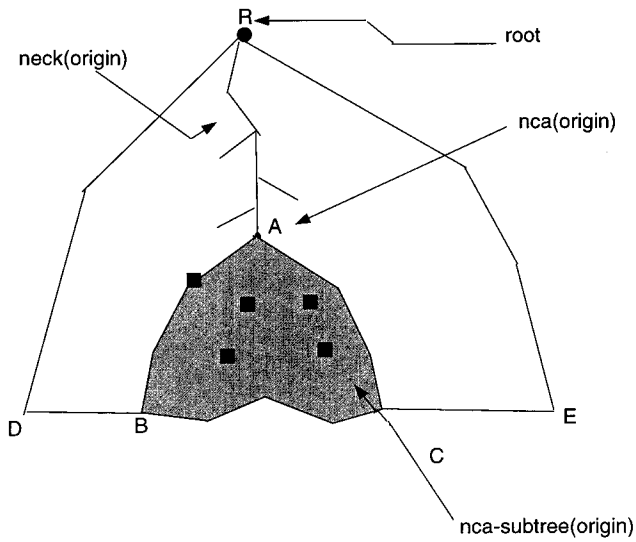

Figure 10. Regions in a binary tree. Illustration of various locations and regions in a binary tree: root, nca(origin), nca-subtree(origin), neck(origin), left(origin), *and* right(origin)

does the possibility of it being a data slice. However, to be useful in debugging one would first test if a smaller tree is a data slice.

Matrices: We consider matrices to be two-dimensional arrays. The dimensions are termed rows and columns. The domain of recursive matrices is one in which the lengths of the two dimensions is not fixed. Intuitively, it implies that a recursive matrix can be defined to contain smaller matrices. We say that a matrix a is a *submatrix* of another matrix b, if $\exists r,c \cdot \forall i,j \cdot a[i,j] = b[r+i,c+j]$ for values where $a[i,j]$ is defined.

The elements of a matrix are indexed on their row and column position. These indices define the absolute position of an element in a matrix. A subset of elements of a recursive matrix may be ordered based on either the row or the column indices. The relative position between pairs of elements may be given as the difference of their row and/or column indices.

If the input domain consists of recursive matrices of variable dimensions, the set of candidates for data slices would be:

(1) The matrix constructed by removing all rows and columns not containing any element in the origin (order dependent).
(2) The matrix constructed by removing all columns not containing any element in the origin (order dependent).
(3) The matrix constructed by removing all rows not containing any element in the origin (order dependent).
(4) The smallest submatrix containing all elements of the origin (relative position dependent).
(5) The smallest submatrix containing all rows containing any element of the origin (relative position dependent).
(6) The smallest submatrix containing all columns containing any element of the origin (relative position dependent).
(7) The smallest submatrix containing the first row of the original matrix and all elements of the origin (absolute position dependent).
(8) The smallest submatrix containing the last row of the original matrix and all elements of the origin (absolute position dependent).
(9) The smallest submatrix containing the first column of the original matrix and all elements of the origin (absolute position dependent).
(10) The smallest submatrix containing a last column of the original matrix and all elements of the origin (absolute position dependent).
(11) The matrices created by treating the original matrix as a fixed structure.

Once again, the size of the candidate matrices, listed above, increases as does the probability of their being a data slice. For the greatest help in debugging, one would test whether a smaller candidate matrix is a data slice first. The set may be pruned by using program-specific knowledge.

5.3. Data slicing by random elimination

When invariance analysis and origin tracking are not applicable or cannot produce small enough data slices, data slicing can be done (or further done) by this method. For recursive structures, we can randomly eliminate some elements from the input data to obtain candidate data slices. For instance, if an $n \times n$ matrix is input to a program, we may randomly drop some rows and columns and check whether it is a data slice by testing the program with it as input.

For fixed structures, we may try to obtain a data slice by replacing some elements in the input by some 'small' elements, such as 0's. The 'small' elements used for replacement are called 'filler' elements.

5.3.1. Example using random elimination

Consider a program that processes an $n \times n$ matrix and fails on the 4×4 matrix given in Figure 11(a). We can randomly eliminate a row and a column, say row 2 and column 3 to obtain a candidate data slice. The candidate data slice is shown in Figure 11(b).

Data slicing by random elimination is most applicable where other methods are not applicable or when a quick attempt at data slicing is desired.

12	45	5	32
−4	78	1	0
345	76	22	79
11	8	21	67

(a) A 4 × 4 matrix causing failure

12	45	32
345	76	0
11	8	67

(b) A candidate data slice - a 3 × 3 matrix

Figure 11. An example of data slicing by random elimination. The candidate data slice is created by eliminating row 2 and column 3 of the original data

5.4. Data slicing by program-specific heuristics

This method creates data slices by using program-specific heuristics most guided by the programmer's intuition. For example, given a finite, linearly-ordered domain and a primitive data element *d* that causes a program to fail, one candidate data slice would be *d*/2, the element half way from the beginning of the domain to *d*. If the program fails for *d*/2, then we have found a data slice. Otherwise, the element half way between *d*/2 and *d* would be the next candidate. The definition of '/', however, may depend on the specific domain. For the set of alphabetic characters, it may be the character half way between 'A' and *d* if *d* is an upper-case character, or between 'a' and *d*, if *d* is a lower-case character.

6. EFFECTIVENESS OF THE DATA SLICING METHODS

In Section 5, several methods for creating data slices have been presented. Since all of the data slicing methods, except invariance analysis, are heuristic, we cannot prove that the methods generate data slices. This section presents a set of empirical studies conducted to assess the effectiveness of the data slicing methods. The empirical studies were performed with four programs acquired from third parties (three from an *ftp* site over the Internet and one from a reference book). These programs are written in C and range in size from 44 to 151 lines of code excluding comments and blank lines.

The empirical studies presented were not controlled, therefore, no significant conclusions can be drawn from them. They only provide a prototype evidence of the utility of our methods in creating data slices. The first author was both the subject and the investigator of the studies. The potential bias resulting from the investigator being the subject as well was reduced by using protocol analysis (Ericsson and Simon, 1984; Shneiderman, 1986). As a subject, he verbalized the protocol of all his actions and rationale and recorded them on an audio cassette. Certain decision actions which were easier to express diagrammatically were logged on hard copy. The protocols were then analysed to draw the inferences reported in this section.

6.1. Methodology

This section presents:

(1) the objective of the studies,

(2) the source of the subject programs used in the study,
(3) the criteria for selecting the programs, and
(4) the procedure followed:
- to induce faults in each program and
- to monitor the activities performed to create data slices.

6.1.1. Objective

The objective of our study was to show that: *the data slicing methods presented can be used to create data slices.*

It was not our objective to assess whether the data slices are indeed helpful in debugging, though from the reduction in processing size, one may infer that the data slice would reduce the debugging time.

6.1.2. Subject programs

Four programs were chosen from a set of 27. Of these, 26 programs were obtained from New York University (made available by Tarak Goradia, currently at Siemens (Goradia, 1993)). Goradia used these programs for his Ph.D. dissertation. One program was taken from a book on C programming (Burkhard, 1988). The collection of programs from NYU (a) spans a diverse application domain, including tax computation, banking, graph algorithms, matrix computation and symbolic algebra, and (b) is available with a suite of test data for each program.

We chose the four programs such that they:

(a) represented different types of input and output domains, i.e., primitive, fixed structure and recursive structures, and
(b) represented programs establishing different types of relationships between the elements of the input and output, namely Types α_i and β_i, $i = 1,\ldots,4$ (see Section 4.2.3).

Table 1 shows the properties of the four subject programs, such as their input, output, the types of relationships between elements of the input and the output, the type of failure, and the data slicing technique used in the study.

6.1.3. Procedure for introducing faults

We were interested in programs which failed on a small set of test cases only (less than one third of the total test cases presented). It made the studies more realistic since the hypothesis is that failures exhibited by large data in production code—code that has been tested and is released for use—are harder to reproduce by smaller data. If a program failed for a majority of test cases, then the problem of reproducing the failure with smaller data would be moot.

Faulty versions of the collection of programs were created by mutating each program (Mathur, 1991; White, 1987; Budd, 1981). To mutate a program means to modify it by

Table 1. Properties of the four programs selected for our study

	Program description	Input domain	Output domain	Types of relationship between input and output	Type of failure	Data slicing technique
1	Strongly connected component	Recursive structure (list)	Recursive structure (list)	$\alpha_2,\beta_2,\alpha_1$	Missing output	I/O analysis
2	Matrix determinant	Recursive structure (matrix)	Primitive	$\alpha_3,\beta_4,\alpha_1$	Incorrect output	Random elimination
3	Change for vending machine	Primitive (integer)	Fixed structure	α_4,β_3	Incorrect output	I/O analysis
4	Bank promotion	Fixed structure (record)	Recursive structure (list)	α_4,β_4	Incorrect output	Origin tracking

applying a predefined function—a mutation operator—such that the modified program is still syntactically correct. Mutation operators include:

- Operator replacement: an operator in a program such as '+', '/', '<', etc. is replaced by another operator.
- Variable replacement: an instance of a variable is replaced by another program variable.
- Constant replacement: an instance of a constant is replaced by another constant.
- Statement deletion: a statement is removed from the program.

Not every mutant created satisfied the criterion that it failed for under one third of the test cases. For instance, some mutants failed for every test case. Those mutants that did not satisfy the criterion were rejected. Therefore, the candidate faulty program was selected after creating several mutants—sometimes by applying more than one mutation to the same program.[1]

6.1.4. Procedure for monitoring data slicing steps

Methods commonly used in *protocol analysis* based studies (Ericsson and Simon, 1984; Shneiderman, 1986) were used to gather information for our study. Tape recordings and hand-written notes were used to record the details of the steps in creating data slices for

[1] Although compound mutations introduce a high level of error complexity, ignoring those did not affect the objective of our studies. Our objective was to show that the data slicing methods we have proposed can be used to create data slices. It was not our objective to study mutation testing.

each program. The recorded information was then analysed and abstracted before presentation in this paper.

6.1.5. Effectiveness of data slicing in experiments

In order to evaluate the effectiveness of the data slicing methods, we measured the reduction in processing size achieved by the data slice. The processing size was measured by instrumenting the program using *btool*—a branch coverage tool intended for software test coverage analysis—to obtain the number of times that various loops (implicit or explicit) are iterated (Hoch, Marick and Schafer, 1990). The reduction of the number of iterations of loops in an execution of the program due to the data slice was used to measure the effectiveness.

6.2. Details of empirical studies

For each case study we present the following:

- a description of the subject program, its input and the expected output;
- a summary observation of the behaviour of the program on our test cases, a description of the data on which it failed and a description of the failure;
- a classification of the program based on the types of its input and output domains;
- details of the steps in creating a data slice from the original failure-causing data;
- a discussion of lessons learned;
- an analysis of the effectiveness of the data slicing methods used.

6.2.1. Case study 1

Program: This study involved program number 24 from NYU. This program finds the strongly-connected components of a directed graph.

Input: The input consists of two parts. The first part consists of just one integer giving the maximum number of nodes in the graph. The second part is a variable length list of edges in the graph, represented as pairs of nodes. A node is represented by an integer between one and the maximum number of nodes.

Expected output: The program outputs the set of connected components in graph. Each connected component consists of a set of nodes. Every node in the graph is expected to be in one and only one connected component.

Observed behaviour: Of the 50 test cases, the program worked correctly for 40. It failed with an input containing eight nodes and 14 edges. The error was obvious in that certain nodes did not appear in the output at all.

Program classification: All elements in the output of this program are of Type β_2 i.e., they are generated from the same element in the input. Similarly, all elements in the second part of the input belong to the Type α_2 in that each generates in the output an element identical to itself. The first part of the input, the number of nodes in the graph, does not generate anything in the output and hence is of Type α_1.

Data slicing details: Given that the input and the output elements of this program are

of Type α_2 and β_2, the origin of the failure elements is explicit. This was important since the failure consisted of absence of some elements in the output. (Hence, its origin cannot be created using dynamic program slicing or abstract interpretation techniques.) Since its input elements are of Type α_2, the failure origin consisted of the nodes in the input that were missing in the output. A subgraph induced by the nodes in the failure origin was the first choice of a candidate data slice. We, however, expanded this subgraph by also including the immediate neighbours of the nodes in the origin. The data slice so obtained was sliced again similarly by considering the missing elements in the output and identifying the failure origin. After several iterations, we created a data slice consisting of just two nodes and one edge. Figure 12 diagrammatically represents the candidate data slices created and the ones found to be data slices.

The original failure-causing input and the final data slice are given in Figure 13.

Lessons learned: Conceptually, the extraction of a subgraph from a graph given a set of nodes in the failure origin sounds simple. However, in practice, given the specific encoding scheme for the graph, this extraction was quite tedious. Since nodes are numbered from one onwards, creation of subgraphs also required renumbering of the nodes. This was a tedious process, one that could have been assisted by an automated tool.

Effectiveness: The program contains six loops, which were iterated a total of 50 times with the initial data. The data slice reduced the total iterations to 14. This is a 72 per

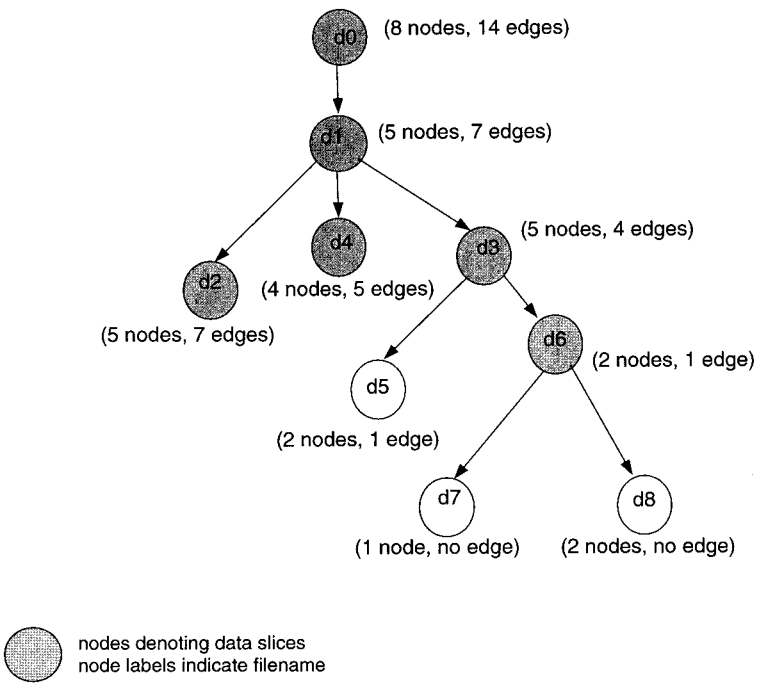

Figure 12. Data slicing by I/O analysis in Case Study 1. The root of the tree is the original faulty input data. The nodes are labelled by the filename which contained the data during the experiment. The nodes are annotated by the number of nodes and edges in the graph. d6 is the final data slice

Original data exhibiting failure	Data slice
8	2
0 1	1 0
0 3	
0 4	
1 3	
1 4	
2 0	
3 2	
4 3	
5 6	
5 7	
6 4	
7 3	
7 5	
7 6	

Figure 13. Original failure-causing input and the final data slice of Case Study 1. The first number gives the number of nodes in the graph. Subsequent pairs of numbers represent edges going from the first node to the second

cent reduction in the total number of iterations, hence also the number of intermediate states. There were two 'for' loops which were each traversed 14 times with the original data, but the same 'for' loops were only traversed once each with the data slice.

6.2.2. Case study 2

Program: This study involved program number 20 from NYU. This program computes the determinant of an $n \times n$ matrix.

Input: It takes as input the dimension of a square matrix. This is followed by the matrix itself with each row of the matrix presented in a single line and a space between pairs of elements of a row. The input, therefore, belongs to the recursive structure domain.

Expected output: The program is expected to output a number corresponding to the determinant of the matrix. The output belongs to the primitive domain.

Observed behaviour: Of the 52 test cases, the faulty program output the correct determinants for 36. It, however, output an incorrect value for the 15×15 matrix given in Figure 15.

Program classification: The number in the output is generated from some computation on the entire matrix. This output, therefore, is of Type β_4. The input consists of the dimension and the matrix. All the elements of the matrix generate one and the same number in the output, hence they are all of Type α_3. The element representing the dimension does not generate anything in the output and hence is of Type α_1.

Data slicing details: The origin of the one and only element in the output is the entire

matrix, since the output element is of Type β_4. Hence, using origin tracking was ruled out. Since the relationship between the input and output for this program is too complex, using I/O analysis was ruled out too. The only choice we were left with was to use random elimination and invariance analysis, of which we chose the former. Candidate data slices were created by randomly eliminating one or more rows and the same number of columns from a matrix causing a failure.

Starting with the 15×15 matrix, we produced the 3×3 matrix, both in Figure 15, where the latter was a data slice of the former. In the course of creating the data slice, 10 other matrices were created, of which six were not data slices and the remaining four were data slices. To find a smaller data slice, nine other 2×2 matrices were created from the 3×3 matrix. None of these matrices recreated the failure. The first four candidates we created, in succession from one another, were data slices, the last one a 5×5 matrix. To derive a 4×4 data slice required generating four candidates, three of which were not data slices. Generating the final 3×3 data slice also required creating four candidates, three of which were not data slices.

In Figure 14, a tree showing the process of data slicing by random elimination in Experiment 2 is given. All the shaded nodes are data slices. The nodes are labelled by test case filename. A tree edge (v1, v2) is labelled by the range of rows and columns which are dropped to obtain v2 from v1.

The original, failure-causing input and the final data slice are given in Figure 15.

Lessons learned: Candidate data slices were created by eliminating rows and columns from another matrix. This was done with the help of a text editor (in this case 'vi'). Though possible, the task was very cumbersome at the initial stages when the matrices were large (15×15, 14×14 and 10×10). Had the initial faulty matrix been larger, say 200×200, as in our case study in Section 2.1, deleting its rows and corresponding columns using a text editor would be a torture. Doing so by a program that edits matrices would be more convenient. If our programs were to be used and supported commercially, we would have built such a program.

Effectiveness: The program contains 10 loops which were iterated a total of 1192 times with the initial data. The data slice reduced the total iterations to 36. This is a 96.9 per cent reduction in the number of iterations, hence also the number of intermediate states. There was one 'for' loop which was traversed 509 times with the original data, but the same 'for' loop was only traversed four times in the data slice.

6.2.3. Case study 3

Program: This study involved a program in a book on C programming (Burkhard, 1988). The program computes change returned by a vending machine.

Input: The input to the program is an integer representing the cost of an item in cents.

Expected output: It outputs the change returned by the vending machine assuming that the customer pays \$10. The change is returned in quarters (25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent). The change is computed assuming that there is no shortage of coins of any denomination. The answer is expected to be unique even though a given amount of money may be returned using more than one combination of change. The uniqueness is achieved by returning a coin of higher denomination whenever possible.

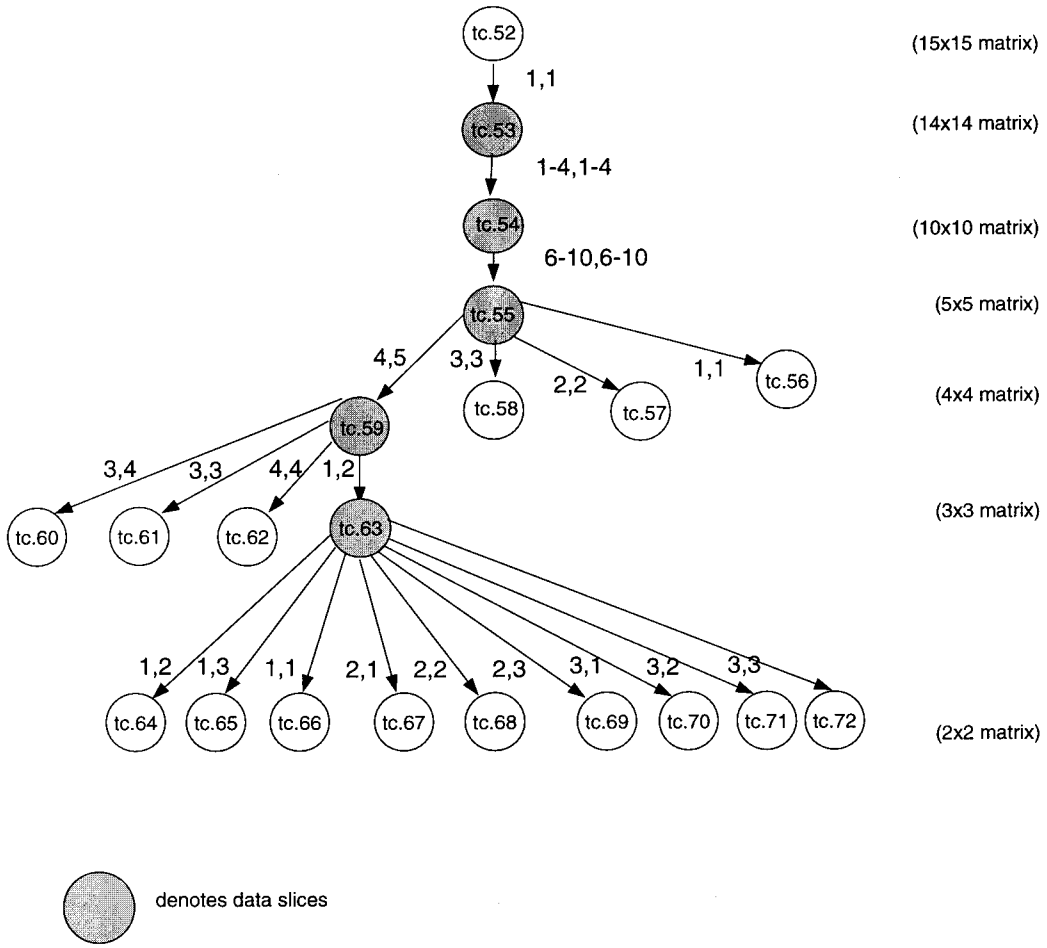

Figure 14. Data slicing by random elimination in Case Study 2. The shaded nodes of the tree are data slices. A node is labelled by the test case filename. The dimensions of node(s) at a level are also given in the diagram within the parentheses. The annotations on the edge indicate the rows and columns deleted to create the next candidate data slice

For example, if the sale is for 950 cents, the change returned would be two quarters amounting to $1000 - 950 = 50$ cents, even though 50 cents may also be returned using five dimes or 10 nickels or 50 pennies, or other combinations of coins.

Observed behaviour: Of the 16 test cases, the program behaved correctly for 11. It produced an incorrect value when the sale amount was 53 cents. Instead of printing '37 quarters, 2 dimes, 2 pennies' it printed '37 quarters, 2 pennies, 2 pennies'.

Program classification: The one and only input to this program is of this Type α_4, since it generates several elements in the output. Similarly, all elements of the output are of Type β_3, since each is generated from this one and only input element.

Data slicing details: In this program, since the input is of primitive type, it would be

The 15 x 15 matrix causing failure:

1.5

0.0 2.0 3.0 -4.0 4.5 2.0 7.0 3.2 12.213 4.341 4.5 6.7 3.7 0.76 5.406

3.0 4.3 4.5 0.5 4.5 2.0 7.0 3.2 12.213 4.341 4.5 6.7 3.7 0.76 1.456

9.85 4.341 4.5 6.7 3.7 0.76 1.456 5.45 2.0 3.0 -4.0 4.5 2.0 7.0 3.2

6.403 2.0 3.0 -4.0 4.5 2.0 7.0 3.2 92.413 4.341 4.5 6.7 3.7 0.76 1.456

1.0 2.0 3.0 -4.0 4.5 2.0 23.0 3.2 21.3 5.091 4.5 7.0 3.7 0.98 1.456

5.4 2.0 3.0 2.1 4.5 0.0 7.0 3.2 12.213 4.341 4.5 7.5 3.7 4.56 155.0

0.0 2.0 3.0 -32.0 4.5 2.0 7.0 5.34 1.2 4.341 4.5 6.7 3.7 5.34 1.456

0.0 4.3 5.4 -4.0 4.5 2.0 5.4 3.2 8.0 7.341 5.43 7.65 3.7 0.76 14.0

0.0 5.09 3.0 -4.0 4.5 2.0 0.0 3.2 13.0 4.341 4.5 6.7 47.0 0.76 56

8.8 0.0 3.0 -4.0 4.5 2.0 8.432 3.2 2.13 0.041 4.5 6.7 53.7 0.76 5.0

1.0 2.0 3.0 -4.0 4.5 0.0 7.0 3.2 12.213 5.2094 4.5 6.7 33.7 0.98 1.6

0.0 4.341 4.32 6.7 3.7 0.76 1.456 1.0 2.0 3.0 -4.0 4.5 2.0 7.0 3.2

4.5 6.7 3.7 0.76 -1.56 1.0 2.0 3.0 -4.0 4.5 2.0 7.0 0.0 12.213 4.31

1.0 2.0 3.0 -4.0 4.5 2.0 4.0 3.2 12.213 4.341 4.5 6.7 3.7 0.76 6.0

1.0 2.0 3.0 -4.0 4.5 2.0 -3.0 3.2 12.213 4.341 4.5 6.7 3.7 0.76 1.0

The 3 x 3 data slice matrix:

3

2.0 5.34 1.2

2.0 3.2 8.0

2.0 3.2 2.13

Figure 15. Original failure-causing input and the final data slice of Case Study 2

in the origin of all the elements in the output, hence origin tracking would not provide any new information.

From the analysis of the incorrect output, it appears that the program prints 'pennies' when it is expected to print 'dimes'. A sale that requires only dimes to be returned as change would be a good candidate data slice.

As a first step we created a sale value so as to not return any quarters. This was done

by adding 37×25 to 53 giving 978. This number was found to be a data slice. The output, however, still contained two pennies. Another data slice was created by adding two pennies to 978 resulting in 980. This too was a data slice, since the program output '2 pennies' instead of '2 dimes'.

The number 980, was the smallest data set for which the program behaved erroneously. It correctly gave the output '1 dime' for the input 990. (Notice that 980 is smaller than 990 with respect to their processing size for this program.)

Hence 980 was the smallest data slice reproducing the failure.

Figure 16 illustrates the data slicing process.

Lessons learned: In this program, a data slice was very easily constructed by input/output analysis. This is because one could easily compute input data that would remove some elements in the output. Unlike previous examples, however, the data slice could not have been generated by editing the data. The data slice 980 could also have been derived just from the observed failure, i.e., without using the original input. The argument goes as follows: 20 cents is the smallest change that can be returned in two dimes and $1000 - 20$ gives the sale for which 20 cents would be returned.

Effectiveness: the program contains two loops which were iterated a total of 45 times with the initial data. The data slice reduced the total iterations to six. This was an 86.7 per cent reduction in the number of iterations, and hence the number of intermediate states. There was one 'while' loop which was traversed 41 times with the original data, but the same 'while' loop was only traversed twice with the data slice 980.

6.2.4. Case study 4

Program: This case study involves program number 19 from NYU. This program evaluates the financial status of a bank's customer and recommends services that may be promoted to that customer.

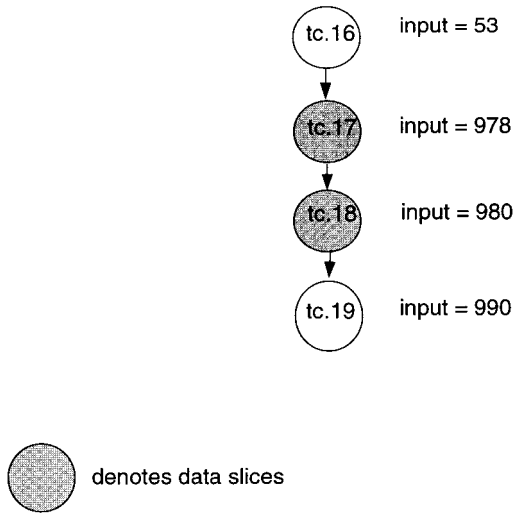

Figure 16. Data slicing in Case Study 3. Test case 'tc.18', integer 980, is the final data slice

Input: A fixed size structure of 15 numbers, each representing a specific information item about the customer's financial status.

Expected output: A list of integers representing various services to be recommended to the customer. The elements of the list are separated by ';'. The recommendations may be for a bank account promotion, a home equity loan, a first home loan or a fixed deposit—encoded using the integers 2, 3, 4 and 5, respectively. When promoting a bank account, the type of account is also given, encoded using an integer.

Observed behaviour: Of the 50 test cases, each containing 15 integers, the program behaved correctly for 39. It failed for one data set in that instead of returning the list '2 3;' recommending promoting a standard bank account, it returned '4; 2 3;', i.e., indicated an additional recommendation to promote a first home loan as well.

Program classification: Since we did not know the exact expected formula used by the program to make the recommendation, we assumed that the elements in the input and output are of Type α_4 and β_4, respectively. However, an element in the output most likely did not depend on every element in the input.

Data slicing details: We wanted to create a data set for which the program incorrectly recommends only a first home loan '4;'. Since the input is a fixed structure, any such data slice will also have 15 elements.

We used origin tracking to find the elements in the input structure used to compute the recommendation '4'. This was an obvious choice because of the hypothesis that the recommendation did not depend on all elements of the input. Besides, since we did not know the actual formula used by the program, I/O analysis and invariance analysis was not possible. The origin of the element '4' in the output consisted of five elements. The first candidate data slice was created by replacing all elements of the input, except those in the origin, by a 0. It turned out that a 0 was an incorrect value for a particular position in the input representing the type of bank account currently held by the customer. This field could take a value ranging from one to four. We tried all the values in sequence and found that the value three led to a data slice. In other words, we created an input with 15 elements for which the faulty program recommended the promotion '4;' whereas the correct version of the program did not make any recommendation. Figure 17 illustrates the process.

The original failure-causing input and the final data slice are given in Figure 18.

Lessons learned: When creating data slices from fixed structures, a simple 'fill all elements not in the origin by 0's' does not seem to be a good strategy. Instead, the values filled in each field should be specific to the information that the field captures.

Effectiveness: This program does not contain any loops, hence any statement in it is either executed once or not at all. The expected reduction in the number of intermediate states is, therefore, insignificant. This is what is found from instrumenting the code. The executions with the original data and the data slice differ in whether a true branch or a false branch of an if statement is taken. This is expected when data slicing a fixed structure.

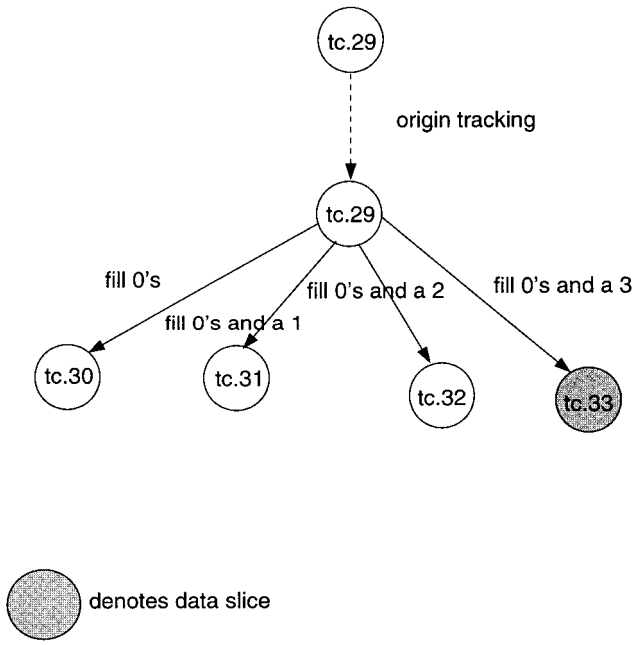

Figure 17. Data slicing in Case Study 4. The nodes are labelled by the test case filenames. The edges are annotated by the operations performed. The dotted edge represents that origin tracking is performed on test case 'tc.29'. The failure origin tc.33 was identified as a data slice

Original data exhibiting failure	Data slice
2 0 2	3 0 0
1137.523364 795.2 999.9	0 0 0
3484.404389 0.249 5988.86421	3484.404389 0.249
10 133 0.5	5988.86421 0.602377
6 470.495234	0 0 0
	0 0

Figure 18. Original failure-causing input and the final data slice for Case Study 4

7. RELATED WORK

7.1. Debugging techniques

Most research effort in debugging has been focused on what we term as *general purpose debugging techniques*. These are techniques, as implied by the name, that may be used in any debugging situation. There are, however, certain debugging situations that are either so frequent or so difficult that practitioners have designed special techniques to cope with them. We refer to these techniques as *scenario-based techniques*. The most

common examples of scenario-based techniques are those designed to debug program failures caused by memory related faults, such as memory leak, uninitialized pointers and dangling pointers. The data slicing techniques presented in this paper are also scenario-based.

We further classify the general purpose debugging techniques on the basis of the 'principles' they are based upon (Chan, 1994). Other classifications of general purpose debugging techniques can also be found in Sevoria (1987), and Ducasse (1992). Our classification is orthogonal to Sevoria's classification of debugging techniques as static and dynamic (Sevoria, 1987). Table 2 provides the classification of various techniques based on our scheme and that due to Sevoria.

None of the static and dynamic general purpose techniques given in Table 2 are effective in debugging when program failures are exhibited by large data. A static technique is obviously not effective since it does not utilize any execution information. A dynamic technique would be effective in such a scenario if it provided a mechanism to reduce the amount of information to be processed in order to gather clues for finding the fault. However, none of the dynamic techniques reduce the number of intermediate states during

Table 2. Classification of various debugging techniques. Each technique is classified based on our operational principle based scheme and Seviora's (1987) scheme

	Seviora's classification	
Operational principles	Static	Dynamic
Cognitive debugging	Pennington and Grabowski (1990) Katz and Anderson (1988)	Pennington and Hastie (1988) Pennington and Grabowski (1990) Katz and Anderson (1988)
Debugging by tracing and inspection		Shimomura and Isoda (1991)
Flow analysis based debugging	Weiser (1984)	Agrawal and Horgan (1990) Korel and Laski (1988) Agrawal, De Millo and Spafford (1991)
Algorithmic debugging		Shapiro (1982) Shahmehri (1991) Fritzson *et al*. (1992)
Debugging by modelling and recognition of errors	Wills (1990) Seviora (1987)	
Origin tracking		van Deursen, Klint and Tip (1993) Dinesh and Tip (1992)
Peritus approach	Pizzarello (1993)	

debugging. They are, therefore, not effective when debugging failures are exhibited by large data (Chan, 1994).

7.2. Origin tracking

The concept of origin tracking was first introduced by researchers in the CWI, Netherlands (van Deursen, Klint and Tip, 1993; Dinesh and Tip 1992). It was developed in the context of term rewriting systems used to specify applications, such as type checkers, evaluators and translators. The inputs and outputs to these applications are terms, as defined in the context of first-order systems. A program in a term-rewriting system reduces an input term to an output term.

An origin of a subterm of the output term, as per the CWI group, is a set of subterms of the input term that is responsible for its creation. In contrast, our definition of origin does not relate all possible subterms, it only relates the 'elements' of the leaf subterms of the input and output term. Our origin relation, therefore, is a subset of the origin relation of the CWI group.

8. CONCLUDING REMARKS

We have observed that the effort required to debug a program depends on the size of the data exhibiting the failure. There are scenarios where the data set exhibiting a failure is large and a smaller data set that reproduces the same problem is desired. Such scenarios demand an inordinate amount of debugging effort as evidenced from the database of debugging anecdotes maintained at the Open University, U.K. (Eisenstadt, 1993). In two extreme cases, reported in this database, the debugging itself was terminated.

In order to help in debugging when the data causing a failure are large, it would help to have another smaller data set, a data slice, that recreates the same failure. The problem of creating such a smaller data set is undecidable. Thus, a smaller data set can only be created by a generate-and-test search over the domain of the program's input. We provide four heuristics—invariance analysis, origin tracking, random elimination and program-specific heuristics—to guide the search for a data slice.

The techniques may be classified on whether or not they depend on the internal structure of the program. Origin tracking by dynamic slicing, origin tracking by abstract interpreation, and invariance analysis depend on the structure of the program. Origin tracking by I/O analysis and random elimination do not depend on the internal structure of the program.

We present the results of several case studies that demonstrate the effectiveness of these heuristics. To be effective, the heuristics presented should be applied in the right situation. Invariance analysis is intended to be used to create data slices for loops. Origin tracking by input–output analysis is good at creating slices for data sets which cause 'missing output' failures. Origin tracking by dynamic slicing or abstract interpretation provides an algorithmic approach to creating data slices though it may be computationally expensive. Program-specific heuristics and random elimination are best applied when other techniques are not applicable or the data slices so obtained are not small enough.

The heuristics we present are amenable to computer assistance and may be incorporated in program development tools. Invariance analysis may be assisted by providing mech-

anisms to test for violation of invariance and mechanisms to save and revert the state of a program. Origin tracking may obviously be automated since it can be done algorithmically. Automated support is also possible for creating candidate data slices from the origin. Such a support may consist of a data structure editor that can perform operations based on absolute position, relative position, order and content of data elements. Since there is no standard representation for data, it may be more prudent to develop a data structure editor generator by enhancing some of the syntax-directed editor. Such a data structure editor may also be useful in data slicing by random elimination. Developing such an automated support is part of our future research agenda.

Acknowledgements

The authors thank the referees for their careful reviews which have helped improve the paper. We thank Nancy L. Franks for carefully typing the manuscript and preparing the figures. The work was partially supported by the grant: Louisiana BoR LEQSF (1993–95) RD-A-38, and the grant: ARO DAAH04-94-G-0334. The contents of the paper do not necessarily reflect the position or policy of the funding agencies, and no official endorsement should be inferred.

References

Agrawal, H., DeMillo, R. and Spafford, E. (1991) 'An execution-backtracking approach to debugging', *IEEE Software*, **8**(3), 21–26.

Agrawal, H. and Horgan, J. R. (1990) 'Dynamic program slicing', *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices*, **25**(6), 246–256.

Budd, T. A. (1981) 'Mutation analysis: ideas, examples, problems, and prospects', in Chandrasekaran, B. and Radicchi, S. (Eds), *Computer Program Testing*, North-Holland, Amsterdam, pp. 129–148.

Burkhard, W. A. (1988) *C for Programmers*, Wadsworth Publishing Company, Belmont CA, 562 pp.

Chan, T. W. (1994) 'Debugging program failures exhibited by voluminous data', Ph.D. dissertation, University of Southwestern Louisiana, Lafayette LA, 110 pp.

Chan, T. W. (1997) 'A framework for debugging', *Journal of Computer Information Systems*, **38**(1), 67–73.

Chan, T. W. and Lakhotia, A. (1993) 'Integrating testing and debugging using program and data slices', in *Proceedings of Computer in Engineering Symposium, ASME 16th Annual Energy-sources Technology Conference and Exhibition*, ASME, New York NY, pp. 1–6.

Chen, T. Y. and Cheung, Y. Y. (1997) 'On program dicing', *Journal of Software Maintenance: Research and Practice*, **9**(1), 33–46.

Dijkstra, E. W. (1976) *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs NJ, 217 pp.

Dinesh, T. B. and Tip, F. (1992) 'Animators and error reporters for general programming environments', Technical Report CS-R9253, Centrum voor Wiskunde en Informatica (CIW), Amsterdam.

Ducasse, M. (1992) 'An extendable trace analyser to support automated debugging', PhD. dissertation, European Computer-industry Research Center, Universite de Rennes I, France.

Eisenstadt, M. (1993) 'Tales of debugging from the front lines', in *Proceedings of the Fifth Workshop on Empirical Studies of Programmers*, Ablex Publishing, Norwood NJ, pp. 86–112.

Ericsson, K. A. and Simon, H. A. (1984) *Protocol Analysis: Verbal Reports as Data*, MIT Press, Cambridge MA, 426 pp.

Fritzson, P., Shahmehri, N., Kamkar, M. and Gyimothy, T. (1992) 'Generalized algorithmic debugging and testing', *ACM Letters on Programming Languages and Systems*, **1**(4), 303–322.

Goradia, T. (1993) 'Dynamic impact analysis: analyzing error propagation in program executions', Ph.D. dissertation, New York University, NY.

Hoch, T., Marick, B. and Schafer, K. W. (1990) *Branch Coverage Tool User's Manual*, University of Illinois at Urbana-Champaign IL.

Horgan, J. and Mathur, A. P. (1992) 'An execution-backtracking approach to debugging', *IEEE Software*, **9**(5), 61–69.
Horwitz, S., Prins, J. and Reps, T. (1988) 'Integrating non-interfering versions of programs', in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Lanugages*, ACM, New York NY, pp. 133–145.
Hutchens, D. H. and Basili, V. R. (1985) 'System structure analysis: clustering with data bindings', *IEEE Transactions on Software Engineering*, **SE-11**(8), 749–757.
Katz, I. R. and Anderson, J. R. (1988) 'Debugging: an analysis of bug-location strategies', *Human Computer Interaction*, **3**, 351–359.
Korel, B. and Laski, J. (1988) 'Dynamic program slicing', Information Processing Letters, **29**(3), 155–163.
Lakhotia, A. and Chan, T. W. (1993) 'Data slicing by dynamic program slicing', in *Proceedings of Software Engineering Research Forum*, University of West Florida, Pensacola FL, pp. 245–253.
Mathur, A. (1991) 'Mutation testing: a tutorial (i–iii)', *SERC Newsletter*, Software Engineering Research Center, Purdue University, W. Lafayette IN, 1st, 2nd and 4th quarter.
Morgan, C. (1988) 'The specification statement', In *On the Refinement Calculus*, Oxford University Computing Laboratory, Oxford, pp. 7–30.
Pennington, A. and Hastie, R. (1988) 'Explanation-based decision making: effects of memory structure on judgement', *Journal of Experimental Psychology: learning, memory and cognition*, **16**(3), 521–533.
Pennington, N. and Grabowski, B. (1990) 'The tasks of programming', in Hoc, J. M., Green, T., Samurcay, R. and Gilmore, D. (Eds), *Psychology of Programming*, Academic Press, London, pp. 45–62.
Pizzarello, A. (1993) A new method for location of software defects, Technical Report, Peritus Software Services, Westboro MA.
Sevoria, R. E. (1987) 'Knowledge-based program debugging systems', *IEEE Software*, **4**(3), 20–32.
Shneiderman, B. (1986) 'Empirical studies of programmers: the territory, paths, and destinations', in *Proceedings of the First Workshop of Empirical Studies of Programmers*, Ablex Publishing, Norwood NJ, pp. 1–12.
Shahmehri, N. (1991) Generalized algorithmic debugging. Ph.D. Dissertation, Linkoping University.
Shapiro, E. Y. (1982) Algorithmic program debugging, An ACM Distinguished Dissertation, MIT Press, Cambridge MA, 232 pp.
Shimomura, T. and Isoda, S. (1991) 'Linked-list visualization for debugging', *IEEE Software*, **8**(3), 44–51.
van Deursen, A., Klint, P. and Tip, F. (1993) 'Origin tracking', *Journal of Symbolic Computation*, **15**(5–6), 523–570.
Weiser, M. (1984) 'Program slicing', *IEEE Transactions of Software Engineering*, **SE-10**(4), 352–357.
White, L. J. (1987) 'Software testing and verification', *Advances in Computers*, **26**, 335–391.
Wills, L. M. (1990) 'Automated program recognition: a feasibility demonstration', *Artificial Intelligence*, **45**(1–2), 113–171.

Authors' biographies:

Tat W. Chan is a faculty member in the Department of Mathematics and Computer Science at Fayetteville State University, Fayetteville, North Carolina. He received his B.S. from the University of Southwestern Louisiana (USL) and his M.S. and Ph.D. degrees from the Center for Advanced Computer Studies, USL. His research interests include Software Debugging and Testing, Software Engineering Education, and Programming Languages. He is a member of the IEEE Computer Society and the Society's Technical Committee on Software Engineering.

Arun Lakhotia is an Associate Professor with the Center for Advanced Computer Studies and the Director of the Software Research Laboratory at the University of Southwestern Louisiana, Lafayette. He received his Ph.D. from Case Western Reserve University, Cleveland, in 1989. His research interests are centred around issues related to improving the productivity of programmers and the reliability of their programs. He is currently directing projects to recover object-orientated design from the source code of legacy systems. The projects are funded by grants from the Defense Advanced Research Projects Agency and the Louisiana Board of Regents. He has also contributed to research in methods and tools for developing logic programs, partial evaluation of logic programs, interprocedural flow analyses and software metrics. Dr. Lakhotia is a member of the ACM and the IEEE.

